From Sequential Programs to Concurrent Threads

Guilherme Ottoni Ram Rangan Adam Stoler Matthew J. Bridges David I. August
Departments of Computer Science and Electrical Engineering
Princeton University

***Abstract*— Chip multiprocessors are of increasing importance due to recent difficulties in achieving higher clock frequencies in uniprocessors, but their success depends on finding useful work for the processor cores. This paper addresses this challenge by presenting a simple compiler approach that extracts non-speculative thread-level parallelism from sequential codes. We present initial results from this technique targeting a validated dual-core processor model, achieving speedups ranging from 9-48% with an average of 25% for important benchmark loops over their single-threaded versions. We also identify important next steps found during our pursuit of higher degrees of automatic threading.**

I. Introduction

For years, a steadily growing clock speed has been relied upon to consistently deliver increased performance for a wide range of applications. Recently, however, this trend has changed, as the microprocessor industry can no longer increase clock speed because of difficulties related to power consumption, heat dissipation, and other factors. Meanwhile, the exponential growth in transistor count remains strong, causing major microprocessor companies to add value by producing chips that incorporate multiple processors. While chip multiprocessors (CMPs) increase throughput for multiprogrammed and multi-threaded codes, they do not directly benefit the many important existing single-threaded applications.

Compiler writers have had little success in extracting thread-level parallelism (TLP) from sequential programs. Good results have been obtained in a few restricted domains, most noticeably in parallelizing scientific and numeric applications [7]. Such techniques perform well on counted loops manipulating very regular, analyzable structures, consisting mostly of predictable array accesses. In many cases, sets of completely independent loop iterations (DOALL) occur naturally or are easily exposed by loop traversal transformations. However, many programs have complex control flow, recursive data structures, or general pointer accesses, rendering these techniques unsuitable in general.

Since automatic thread extraction has been difficult for compiler writers to achieve, computer architects have turned to speculative [3], [6], [10] and multiple-pass [8], [2] techniques to make use of additional hardware contexts. These techniques are promising, but generally require significant hardware support to handle recovery in the case of mis-speculation or to affect the warming of microarchitectural structures. These approaches are also limited by the increasing mis-speculation rates, penalties, and pollution encountered as they become more aggressive. Even the best of these techniques do not replace the need for automatic, non-speculative thread extraction. Instead, they play an important, largely orthogonal role.

The goal of this work is to automatically thread ordinary C programs, inspired by the above mentioned compiler and computer architecture work and by successes of ILP-extraction techniques. Three obstacles become apparent. Overcoming the primary obstacle involves embracing a type of parallelism that is easier to find in sequential applications, *pipelined parallelism*. Doing so implies relatively minor hardware changes, but allows leveraging of powerful ILP optimizations and analyses to extract TLP. The two secondary obstacles involve limitations in today's compilers that we hope to address in the future.

This paper describes these obstacles and our approach to eliminating them. We then describe a compiler technique, built on an aggressive ILP compiler, that overcomes the primary obstacle by automatically extracting, without resorting to speculation, long-running, concurrently-executing, pipelined threads from unmodified sequential C programs. We also briefly describe the hardware support necessary to execute these pipelined threads without costly synchronization overhead. Using a dual-core processor model with validated processor cores, we demonstrate promising initial results.

II. Obstacles to Thread Extraction

A. Type of Parallelism

Significant untapped parallelism already exists in sequential applications. Due to complex control flow and irregular pointer-based memory accesses, this parallelism is not of the DOALL type at which scientific parallelization techniques excel. Instead, our limit studies show that loops in sequential C/C++ codes generally have one or more cross-iteration dependence chains. Fortunately, in such cases, parts of each iteration can be pipelined and overlapped with other sections in different iterations. This *pipeline parallelism* is often exploited by ILP techniques such as loop unrolling and software pipelining (but often with varied success in this domain due to variable latencies). However, since these techniques do not extract threads, they cannot be directly applied to chip multiprocessors. Pipelined parallelism is also exploited with DOACROSS thread parallelization, but such techniques are not general enough to handle uncounted loops, control flow and irregular pointer-based memory accesses [7]. Additionally, in creating DOACROSS loops, the inter-core communication latency is inserted on the recurrence of the loop, elongating the critical path of the loop in the amount of the number of iterations multiplied by the communication latency. The technique presented in this paper exploits pipelined parallelism without such limitations. In particular, no communication latency is inserted on loop critical paths.

Manuscript submitted: 3 May 2005. Manuscript accepted: 13 June 2005. Final manuscript received: 20 June 2005.

Posted to the IEEE & CSDL on 1/30/2006
DOI 10.1109/L-CA.2006.5

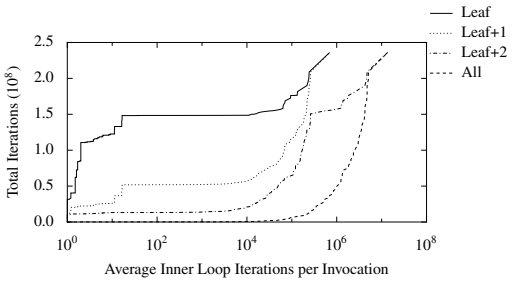


Fig. 1. For an aggressively inlined 186-crafty, the trip-count of loops visible within each function ("Leaf"); within each function and some number of parents ("Leaf+N"); and across the whole program ("All") as a cumulative distribution function weighted by total inner-loop iteration count.

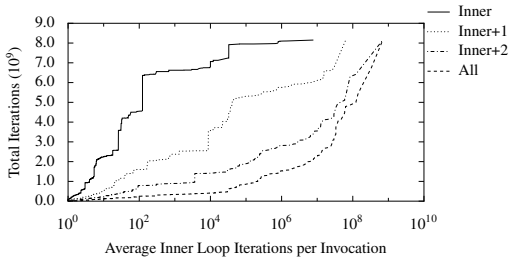


Fig. 2. For SPECINT 2000 programs, the trip-count for inner loops ("Inner"), for some level of loop nest ("Inner+N"), and for all levels ("All") as a cumulative distribution function weighted by total inner-loop iteration count.

B. Scope of Optimization

Extracting pipeline parallelism involves the partitioning of loops in some fashion. Naturally, to be profitable, the loops partitioned must represent a significant portion of the total execution time and must be long running (many iterations per invocation) to overcome any one-time costs. Loops of this type generally exist in programs, but are not always visible to the compiler. Since traditional compilers use functions as the unit of optimization (though analysis may be inter-procedural), inlining is employed to increase the scope of optimization [5]. Full inlining is not possible because inlining increases code size beyond acceptable limits and cycles may exist in the call graph. As a result, profitable loops may not be visible to the compiler for TLP extraction even after aggressive inlining.

Consider Figure 1, which characterizes an aggressively inlined version of 186.crafty from SPECINT2000. For each inner loop in the benchmark, the iterations were assigned to the outermost loop(s) in the function. The outermost loop(s) was then categorized according to the number of iterations per invocation (IpI). Each line in the figure is a cumulative distribution of the total number of inner loop iterations for all outermost loops with IpI at most the given x-axis value. The solid line labeled "Leaf" shows that, when restricting visilibity to leaf functions after aggressive inlining, roughly two-thirds of total program iterations occur within loops which iterate less than 20 times per invocation. Generally, only loops with an IpI of at least 10^3 are suitable for optimization. Thus, limiting the compiler's scope to a single function is unlikely to yield much gain for optimizations with one-time costs. The remaining lines show that the situation improves if compilation scope were to include either 1, 2, or all calling functions when determining outermost loops. Not surprisingly, when the outermost loop(s) is the main loop of the program, the IpI is a significant fraction of the program runtime. To increase program optimization scope, we are currently exploring an alternative to inlining which gives compilers whole-program scope without code growth.

Just as Figure 1 shows that inlining is not enough, Figure 2 shows that limiting partitioning to just inner loops, even when ignoring function scoping, also leads to reduced optimization opportunities. Roughly half of the innermost loops, weighted by inner loop iteration count, have trip counts less than 100, as shown by the *Inner* line. The remaining lines show that assigning the innermost iterations to the loop that contains the innermost loop greatly increases the IpI. This is consistent with our experience, which has shown that inner loops are not sufficient for optimization. As a result, the technique presented in this paper is designed to handle arbitrary control flow, including control flow created by nested loops.

C. Memory Analysis

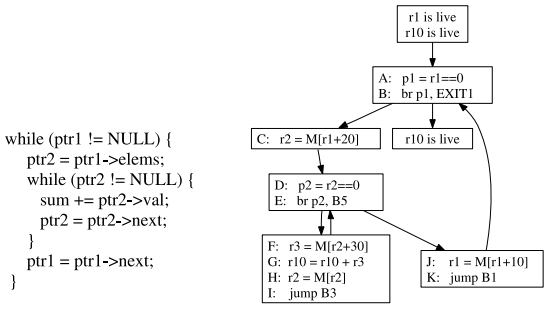
With the goal of extracting ILP, researchers have produced exceptional memory dependence analysis techniques. These techniques are sufficient to enable the successful extraction of threads as described here. However, we observe two problems that, if addressed, would produce better results[1]. First, since we extract threads from low-level codes (for reasons described in the next section), memory analysis information must be accurate in the back-end of the compiler. Traditionally, compilers perform memory analysis in the front-end and propagate this information to the back-end. However, conservative propagation during optimizations degenerates the accuracy of this result [4]. Second, since sequential codes often have recursive data structures, a shape analysis scalable to real codes would reduce false cross-iteration dependences.

III. Decoupled Software Pipelining

In our approach, we leverage prior successful ILP compilation research to address the compilation challenge of Section II-A. We illustrate our approach using the code of Figure 3, which sums the elements of a linked list of linked lists of integers. Figure 4(a) shows the dependence graph corresponding to the code in Figure 3(b). This dependence graph accounts for all relevant dependences: data, control, intra-iteration (solid), and loop-carried (dashed lines).

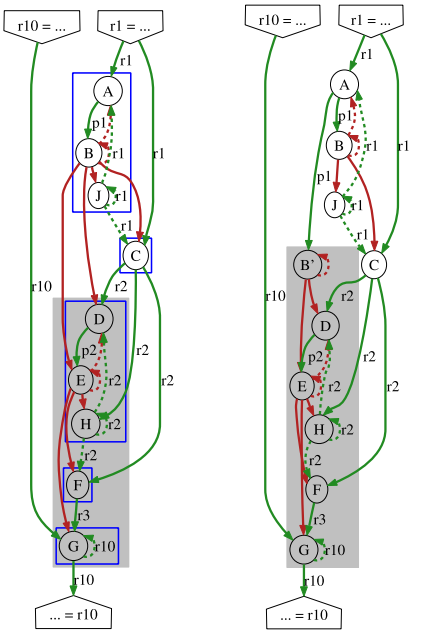
Like the loop in Figure 3, loops typically found in sequential codes are not DOALL in nature. DOACROSS partitioning is not profitable due to reasons mentioned earlier. Our technique, called *Decoupled Software Pipelining* (DSWP), is a more general form of *pipeline parallelization* that overcomes the drawbacks of DOACROSS parallelization (1) by working in the presence of any kind of memory dependences; (2) by extracting more scalable parallelism; and (3) by reducing sensitivity to inter-thread communication latency (by keeping it out of the critical path). DSWP was first introduced as

[1]These problems manifest, for example, in *epicenc*, our lowest performing benchmark. The 9% gain becomes 45% without these problems.



(a) C source code (b) Intermediate representation

Fig. 3. Sample code. Adds up the elements in a list of lists.



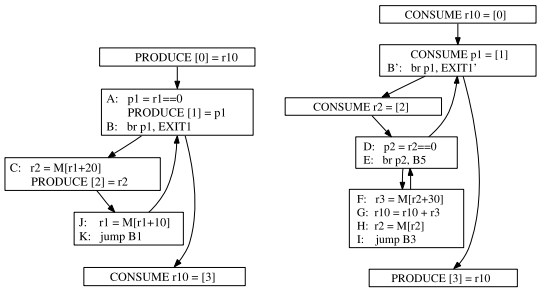
(a) Initial (b) Node-split B

Fig. 4. Dependence graphs. Data dependences are labeled with the corresponding data item. Control dependences are unlabeled. Intra-iteration dependences are solid, and loop-carried dependences are dashed.

a method to hide cache misses in the traversal of recursive data structures as illustrated in prior work using hand-modified benchmarks [9]. Here, we establish DSWP as a more general thread parallelization technique and propose a systematic way to perform it in a compiler.

In order to support the pipelined communication of values between threads, DWSP uses a *synchronization array* to provide a conduit for communication and synchronization between the threads [9]. The synchronization array can be a special register file that implements queue semantics between cores. The threads execute `PRODUCE` and `CONSUME` instructions to send and receive values using the queue. These instructions, which stall only when the queue is full or empty as appropriate, are the only necessary extensions to the ISA.

DSWP transforms a loop into a sequence of threads, forming a pipeline of threads wherein each thread acts as a pipe-stage performing part of the computation of the original loop. The new threads execute concurrently and are long running, each generally consisting of some loop or loop nest. Forming a pipeline requires that dependences only flow unidirectionally among the set of threads. To ensure this flow, the compiler first identifies the strongly connected components (SCCs) in the dependence graph, as illustrated by the rectangles in Figure 4(a). After this, DSWP uses a load-balancing heuristic to decide where to partition the dependence graph. In the example of Figure 4(a), the instructions outside the shaded rectangle are assigned to the first or main thread (also running the rest of the program), and the instructions inside this rectangle are assigned to the second thread. The arcs that cross the boundaries of this rectangle correspond to values that need to be communicated between the threads. This communication is implemented by passing values through the synchronization array. In order to allow control dependences to be communicated (the unlabeled arcs from `B` to `C`, `D`, and `E` in the second thread), the branch instruction `B` is node split. Node `B` in Figure 4(a) becomes `B` and `B'` in Figure 4(b)). This node splitting ensures that control dependences are communicated as data dependences. Figure 5 illustrates the resulting code for this partitioning, with the synchronization instructions inserted. Note that inside the loops, the values are only transmitted in one direction. This implies that the communication latency is a one-time cost, not a recurrent one. In addition, initialization (and finalization) instructions are necessary before (and after) the loops, in order to pass loop *live-in* (and *live-out*) values between the threads.



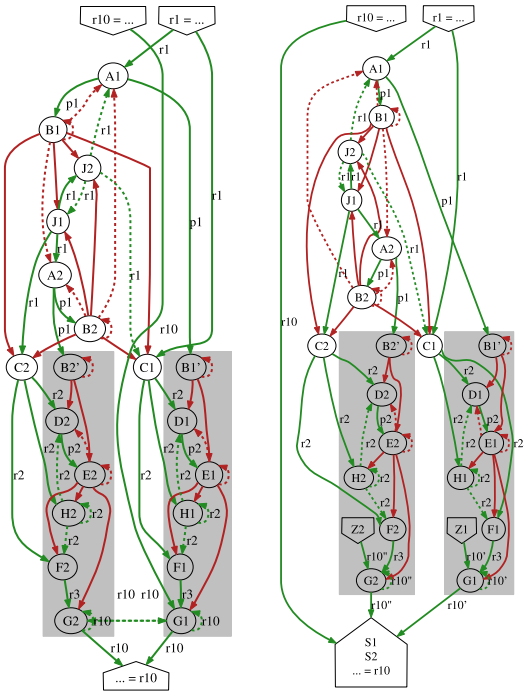
(a) Main, producer thread (b) Consumer thread

Fig. 5. Two thread partitioning. The main thread is also the producer.

We have created a proof-of-concept implementation of DSWP using the IMPACT compiler [1], targeting a dual-core Itanium 2 model with the synchronization array. Our simulator model was built using the Liberty Simulation Environment [11], and each core has an accuracy to within 6% of the corresponding real hardware. Table I presents the results of applying DSWP to a set of benchmarks (mostly from the SPEC2000 benchmark suite). For each benchmark, we chose the loop that corresponds to the largest share of the total execution time (between 16% and 90%). The average speedup obtained is 25%, with a range of 9-48%. The next section presents several factors that can potentially improve these results and lead to partitioning of more than two threads.

TABLE I
LOOP REPRESENTATION (IN % OF EXECUTION TIME) AND SPEEDUP.

Benchmark	Representation %	Speedup %		Dyn. Instr. Increase %
		Loop	Overall	
wc	90	47	40.4	12.2
epicdec	29	9	2.5	17.7
jpegenc	20	12	2.2	20.2
129.compress	16	10	1.5	23.8
179.art	21	24	4.2	78.5
181.mcf	35	48	12.8	64.9
183.equake	67	44	25.7	20.6
188.ammp	64	9	5.6	6.1
Average	42.8	25.4	11.9	30.5
Geo Mean		24.3	11.1	28.4
Standard Dev.		18.0	14.1	26.2



(a) Unroll producer, split consumer

(b) Accumulator expansion

Fig. 6. Dependence graphs for transformations to get more threads.

Table I also shows the increase in the number of dynamic instructions for DSWP over the base.

IV. FUTURE WORK

The first avenue of future exploration is the use of a more accurate memory analysis technique as discussed earlier, building on [4]. In addition, other optimizations (e.g. accumulator expansion, induction variable expansion, and expression reformulation) will be applied to reduce the dependences among instructions, creating more SCCs. Optimizations to reduce inter-thread communication bandwidth are also of interest.

While only two threads are extracted by the technique as presented, additional threads can be obtained in several ways. First, more stages can be created in the pipeline. Second, DSWP exposes parallelism in the form of a *DOALL consumer*. Figure 6(a) illustrates the dependence graph for the code obtained by unrolling the original loop. The shaded region in Figure 4(b) becomes two regions in Figure 6(a). This could be split into two consumer threads if not for the loop-carried dependences between `G1` and `G2`, which add to the accumulator register `r10`, preventing the transformation. Accumulator expansion can remove this dependence by using a different register in each thread and adding them at the exit of the loop. The dependence graph after applying this transformation is shown in Figure 6(b), and it can be used to infer the resulting threads. This technique can also be used to obtain an arbitrary number of consumer threads.

V. CONCLUSION

This paper explores the problem of extracting thread-level parallelism from sequential programs for execution on modern CMPs. We described the main challenges of this problem and our approach to addressing them. The key insight involves partitioning for easier-to-find *pipeline parallelism* at the instruction-level, effectively exploited by the *Decoupled Software Pipelining* technique. Initial results show speedups on average of 25% and as high as 48%, across important benchmark loops. Future work involves increasing the scope of compiler optimization, improving the accuracy of low-level memory analysis, and applying ILP compiler techniques to this new form of TLP partitioning.

REFERENCES

[1] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu, "Integrated predication and speculative execution in the IMPACT EPIC architecture," in *Proc. of Int'l. Symp. on Computer Architecture*, 1998.

[2] R. D. Barnes, E. M. Nystrom, J. W. Sias, S. J. Patel, N. Navarro, and W. W. Hwu, "Beating in-order stalls with 'flea-flicker' two-pass pipelining," in *Proc. of the Int'l. Symp. on Microarchitecture*, 2003.

[3] A. Bhowmik and M. Franklin, "A general compiler framework for speculative multithreading," in *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*, 2002, pp. 99–108.

[4] B. Guo, M. J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D. I. August, "Practical and accurate low-level pointer analysis," in *Proc. of the Int'l. Symp. on Code Generation and Optimization*, 2005.

[5] W. W. Hwu and P. P. Chang, "Inline function expansion for compiling realistic C programs," in *Proc. of ACM Conf. on PLDI*, 1989.

[6] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar, "Min-cut program decomposition for thread-level speculation," in *Proc. of ACM Conf. on Programming Language Design and Implementation*, 2004, pp. 59–70.

[7] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures*. Morgan Kaufmann Publishers Inc., 2002.

[8] D. Kim and D. Yeung, "A study of source-level compiler algorithms for automatic construction of pre-execution code," *ACM Trans. Comput. Syst.*, vol. 22, no. 3, pp. 326–379, 2004.

[9] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August, "Decoupled software pipelining with the synchronization array," in *Proc. of Int'l. Conf. on Parallel Architectures and Compilation Techniques*, 2004.

[10] J.-Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.-C. Yew, "The superthreaded processor architecture," *IEEE Transactions on Computers*, vol. 48, no. 9, pp. 881–902, 1999.

[11] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August, "Microarchitectural exploration with Liberty," in *Proc. of Int'l. Symp. on Microarchitecture*, 2002, pp. 271–282.